Trustworthy Computing

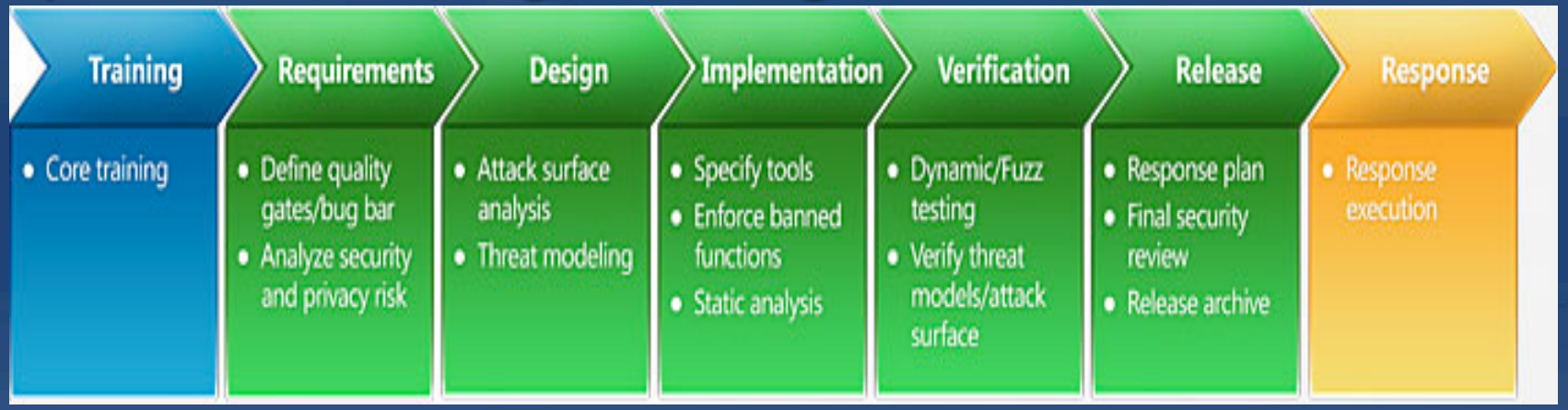
Effective Fuzzing Strategies

Jason Shirk
Program Manager - Fuzzers
Security Science
Microsoft Security Engineering Center (MSEC)

Microsoft

Security is a journey, not a destination

- The Security Development Lifecycle (SDL) was created to build security into the process of engineering software



- Part of the process is verifying that the security measures in place actually work
- Fuzzing can be used for verification

Answer These Questions First

- How do I know if my fuzzing is effective?
- Have to answer 3 other questions first
  - What approach should I take?
  - What do I look for when I fuzz?
  - How much is enough?
- Then an effective strategy can be built

WHAT APPROACH SHOULD I TAKE?

What Approach Should I Take?

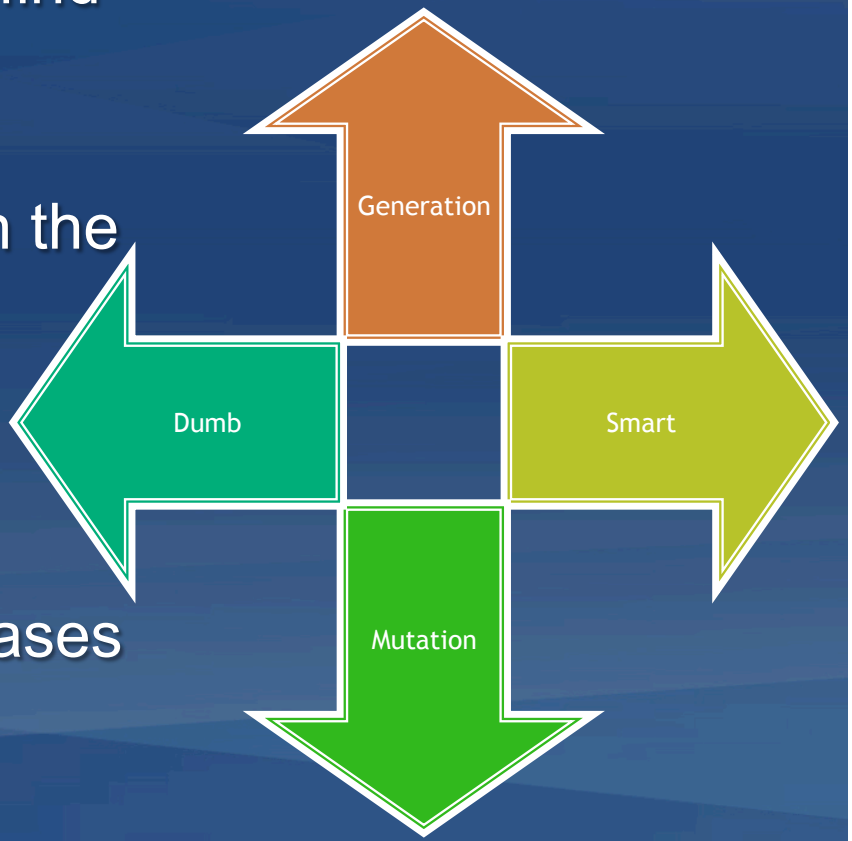
- Defining the Target
- What are the Tools?
- What's Most Effective?

Define the Target

- What are you really fuzzing?
  - Web Service
  - Protocol Parser
  - File Parser
  - Local Service
- What Type of Data is Being fuzzed?
  - Binary
  - Text
- Are there Layered Attack Surfaces?
  - Is there a wrapper?
  - Is it compressed?
  - Is there initial validation that would reject fuzzed data?

What are the Tools?

- Dumb Fuzzers
  - Easy to build and easy to use
  - Relatively low-investment to find a lot of bugs
  - Penetration may not be very deep
  - Preferred method by many in the industry
- Smart Fuzzers
  - High cost of entry
  - Format aware
  - Highly configurable
  - Better penetration in some cases
  - Find different bugs
- Generation vs. Mutation



About Dumb Fuzzing

- Dumb fuzzers are (mostly) not format aware, and just flip bits
  - Sequential fuzzers
    - Start with a value, and increase/decrease from that value until done
    - e.g. 0x00, 0x01, 0x02, …0xFE, 0xFF
  - Random Bit-flipping
  - Random String/Binary Injection
- Still very effective with great ROI

Tool tip: Peach and MiniFuzz both do dumb fuzzing

Smart Fuzzing Case Study

- MS07-017 had to do with repeating ANI headers
  - 1st ANIH ☺
  - 2nd ANIH ☹
  - Wrapped by an Exception Handler
- Have to fuzz the framework and not just the values
- A dumb fuzzer would never find this issue
- A smart fuzzer could find it
  - If the grammar is too strict, it wouldn't fuzz the headers and could miss this type of issue
- The debugger has to be smart enough to catch first chance exceptions

Tool Tip: Peach does smart fuzzing too

Fuzzing at Microsoft

- We Use Semi-Dumb Mutational Fuzzers First
  - Mutating existing data gives us a high probability of the fuzzed data being accepted by the target
- Developing custom Smart fuzzers has not provided a very good ROI
  - Smart Fuzzing takes a lot of domain knowledge to build
  - Smart Fuzzers are typically complicated to configure
- Research has led us to use a combined approach – Illustrated by the Fuzzing Olympics
  - Providing a minimal amount of initial “Fix-up” in a script is much easier than trying to define a type (e.g. CRC’s)
  - Dumb Fuzzers are easy to deploy

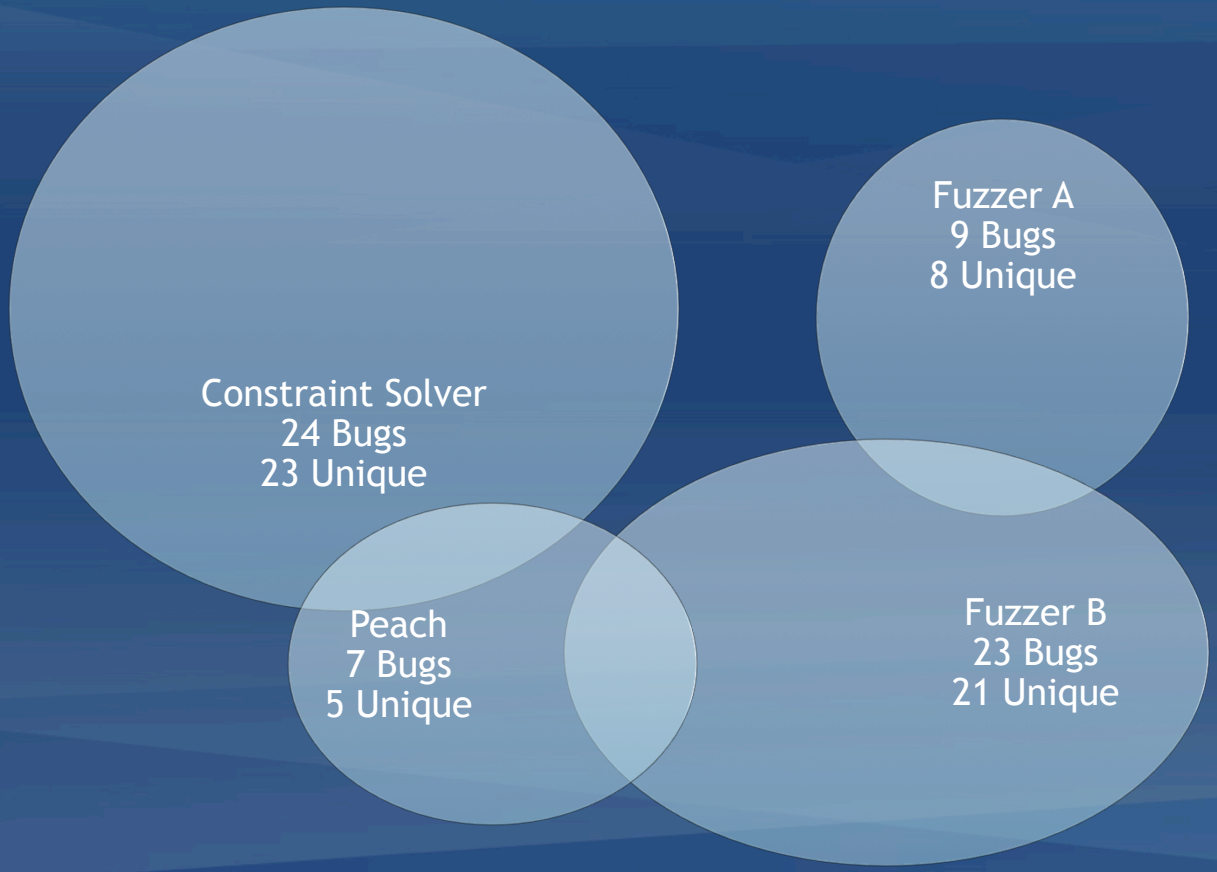
Microsoft Fuzzing Olympics

- Several tools competed head-to-head
  - Several Internal Mutational Fuzzers
  - A Constraint Solver
  - Peach – An External Mutational/Generation
- Level playing field
  - Same timeframe
  - Same targets
    - 1 text parser, 1 binary parser – both previously untested

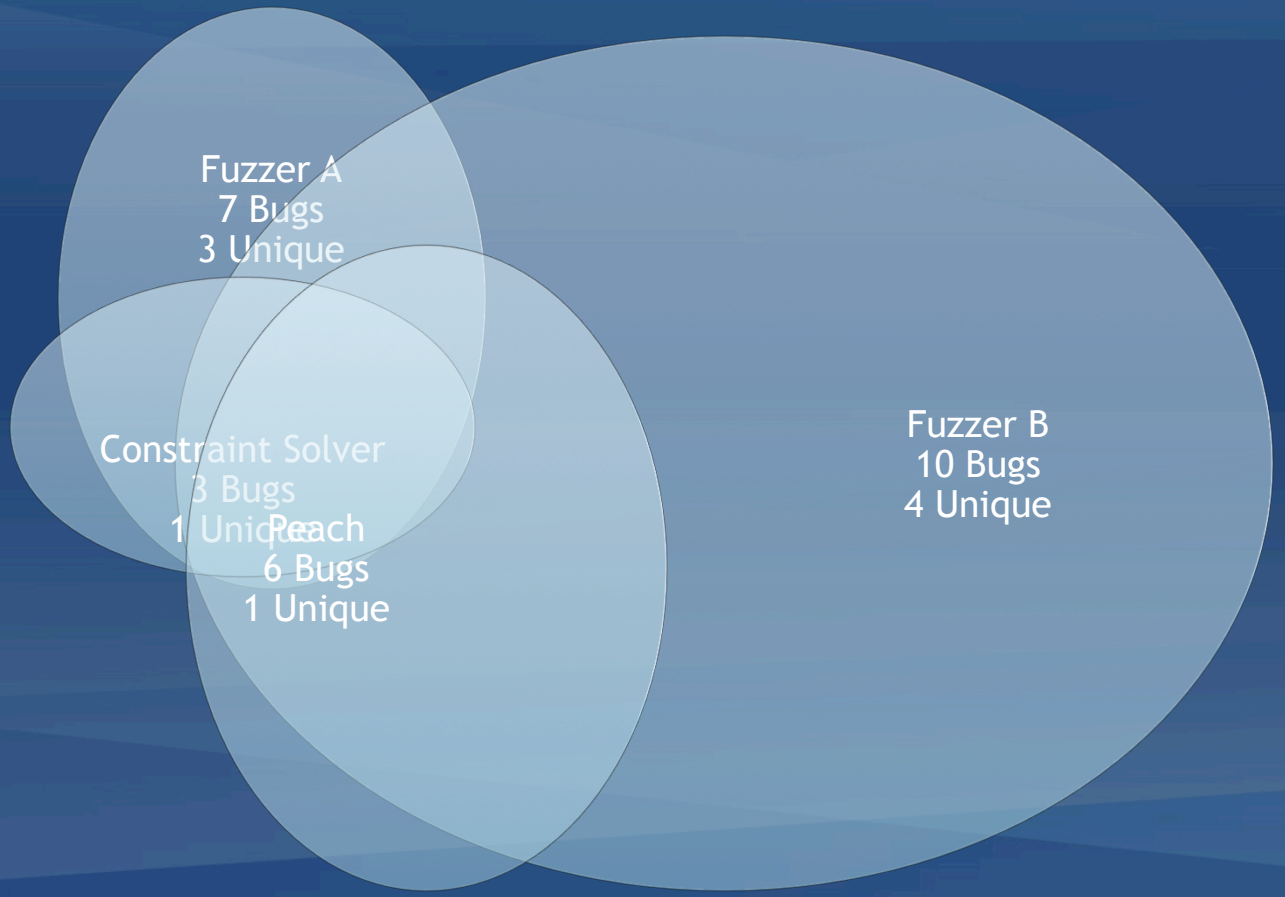
Olympics Findings

- There was a lot of overlap in the bugs found
- Some fuzzers did much better at binary vs. text formats
- No one fuzzer found ALL of the bugs
- A lot of bugs were discovered, including one definitive MSRC grade issue
- Many of the bugs found were in close proximity to others in the code (major hashes)
- Developing custom file descriptions did not appear to provide a very good ROI
  - Dumb Fuzzing found the majority of the bugs in the Olympics
  - Other internal fuzzing efforts support this as well
- Automated Format Analysis found more bugs (minor hashes) than any other fuzzer, but it's more complicated than that…

Olympic Results – Text Parser
60 distinct crashes



Olympic Results – Text Parser
10 underlying bugs



Diversify!

- A Primary Rule of Fuzzing:
  - Change your approach, find different bugs
- Try a different method
  - Mutational
  - Generation
  - Sequential
  - Constraint Solving
- Fuzzing with a second approach measurably increased effectiveness
  - 10%-300% in this case

Make the Most of Your Tools

- Check for penetration
  - Validate code coverage
  - Consider bypassing or proxying any tricky authentications, and test those separately
- Create custom fuzzers for small hard-to reach areas
- Template Reduction (Mutation only)
  - Using the smallest number of templates with the maximum amount of Coverage
  - Template Reduction increases effectiveness by ~100%*

*Based on research by Gavin Thomas, MSRC Engineering

WHAT DO I LOOK FOR WHEN I FUZZ?

Scaling a Difficult Problem

- Problems exist with identifying unique crashes
  - The same issue can arise multiple times
  - The same issue can arise through multiple code paths
  - The same issue can be found across multiple machines
- Classifying the crashes is another issue entirely
  - Manual inspection of crash dumps does not scale
  - Identifying security issues takes experienced resources
  - Takes a lot of time to manually analyze the crash
- Testing produces more crashes than there are resources to triage
  - Automation can help trim down the triaging
  - Grouping crashes by location in code helps

Requirements for a Solution

- Performance in Live Debugging
  - Security tools have to evaluate every first chance exception for security implications
- Minimize False Positives
  - Tools must scale, noise blocks that
- Avoid False Negatives
  - Categorizing an exploitable crash as not being exploitable is a major fault
- Maintainability and Expandability
  - The state of the art in exploitation changes, tools must keep up

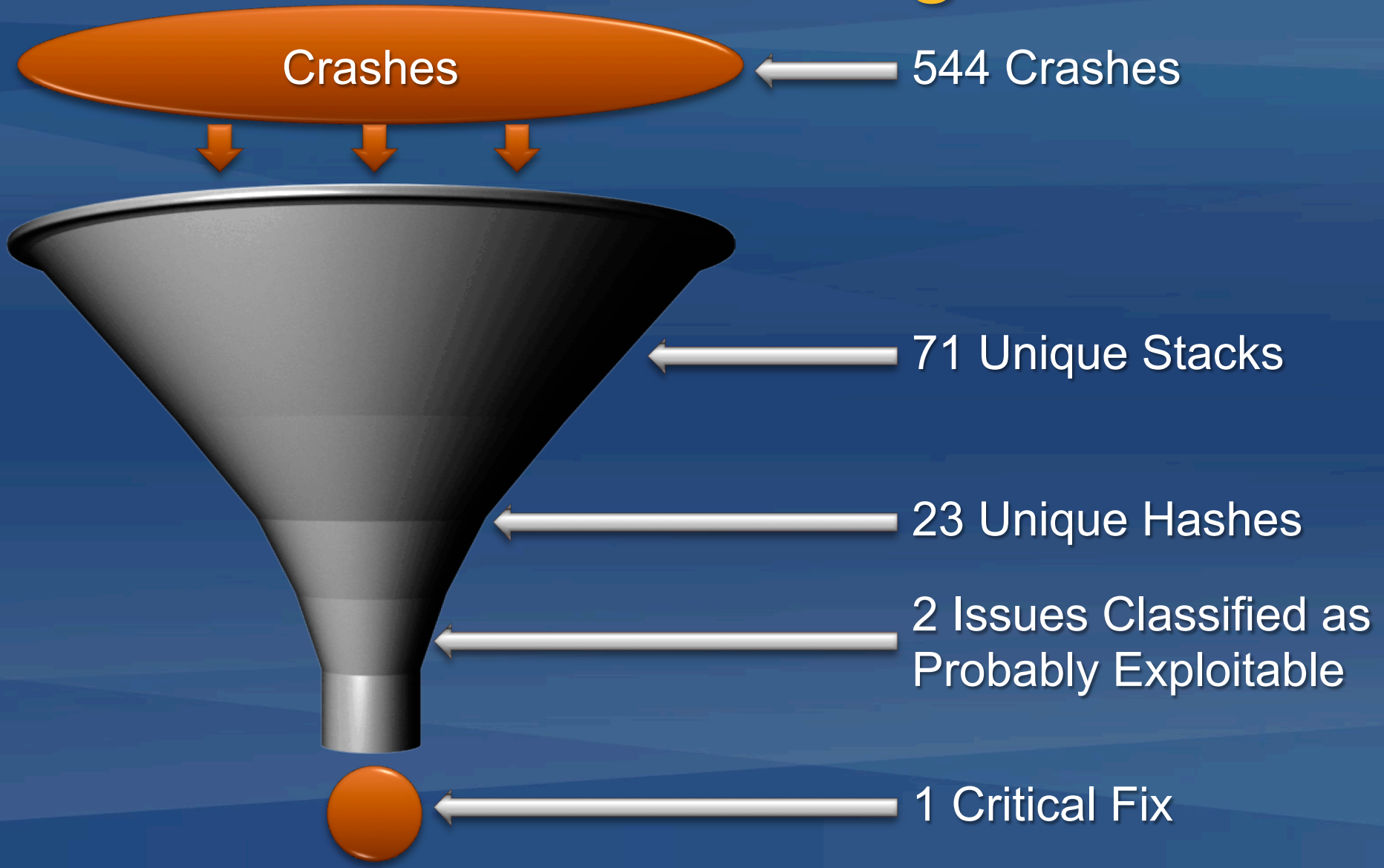
Architecture for a Solution

- Rules Driven Engine
  - Maintainability
  - Ease of understanding the rules
- Processor independence
  - The prototype already had too many special cases for x86 versus x64 cluttering the code flow
- Human and Machine readable output
  - Other tools will be layered on top
  - Manual use is still a supported scenario
- Implemented as a Windows® Debugger extension (windbg.exe)

!exploitable Crash Analyzer

- What is it?
  - Windows debugger extension (Windbg.exe)
  - Provides automated crash analysis
  - Provides security risk assessment
- How does it work?
  - A live crash or dump is created by a debugger on Windows
  - !exploitable examines crash data
  - Identifies the uniqueness of each crash
  - Analyzes data to provide reliable guidance on exploitability
- What is the output? (Bucketizing)
  - An exploitability indicator identifies whether the crash is:
    - Exploitable
    - Probably Exploitable
    - Probably Not Exploitable
    - Unknown
  - A set of identifying uniqueness indicators
    - Hashes

Automated Crash Triage



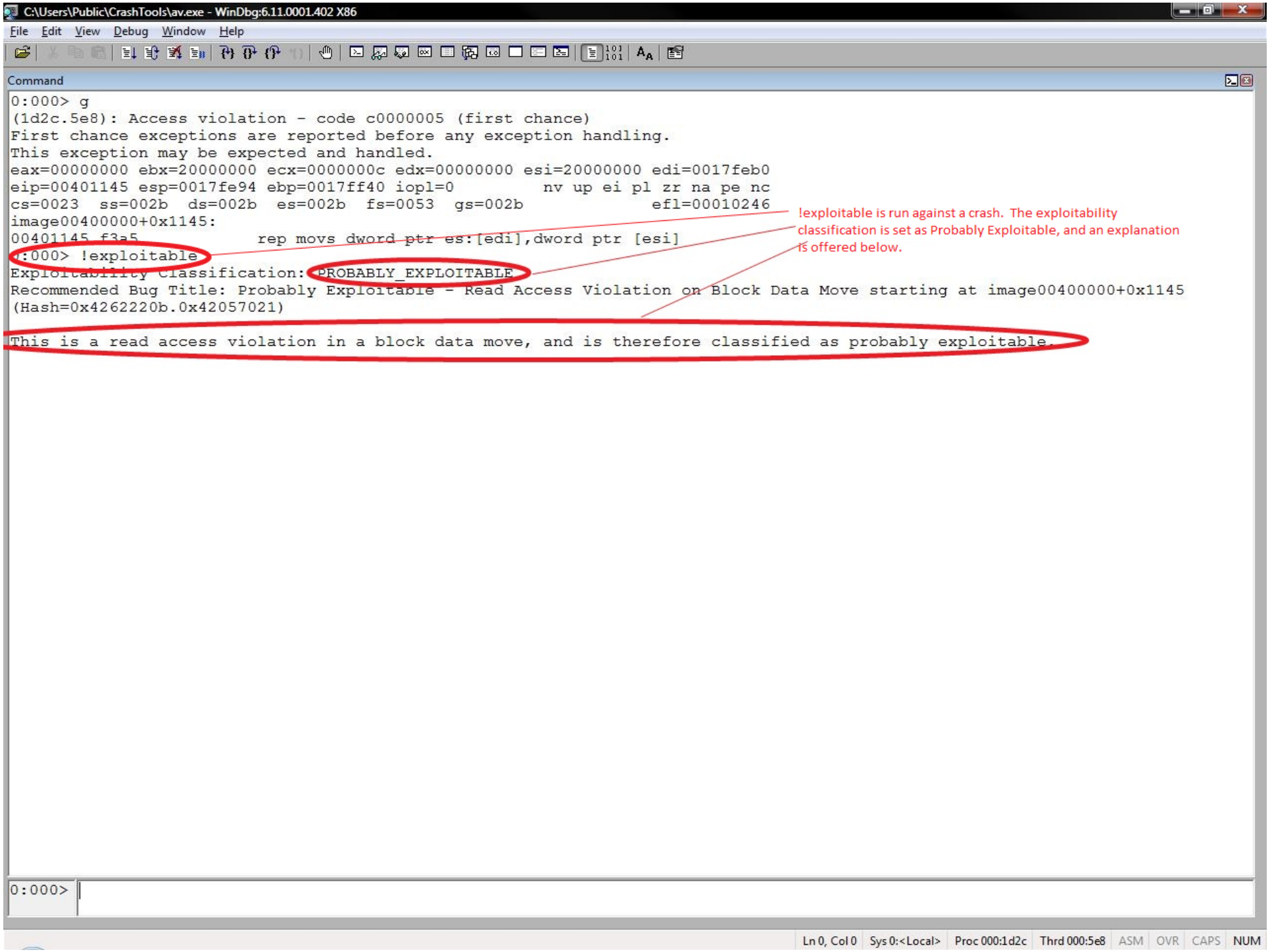
Assumptions and Limitations

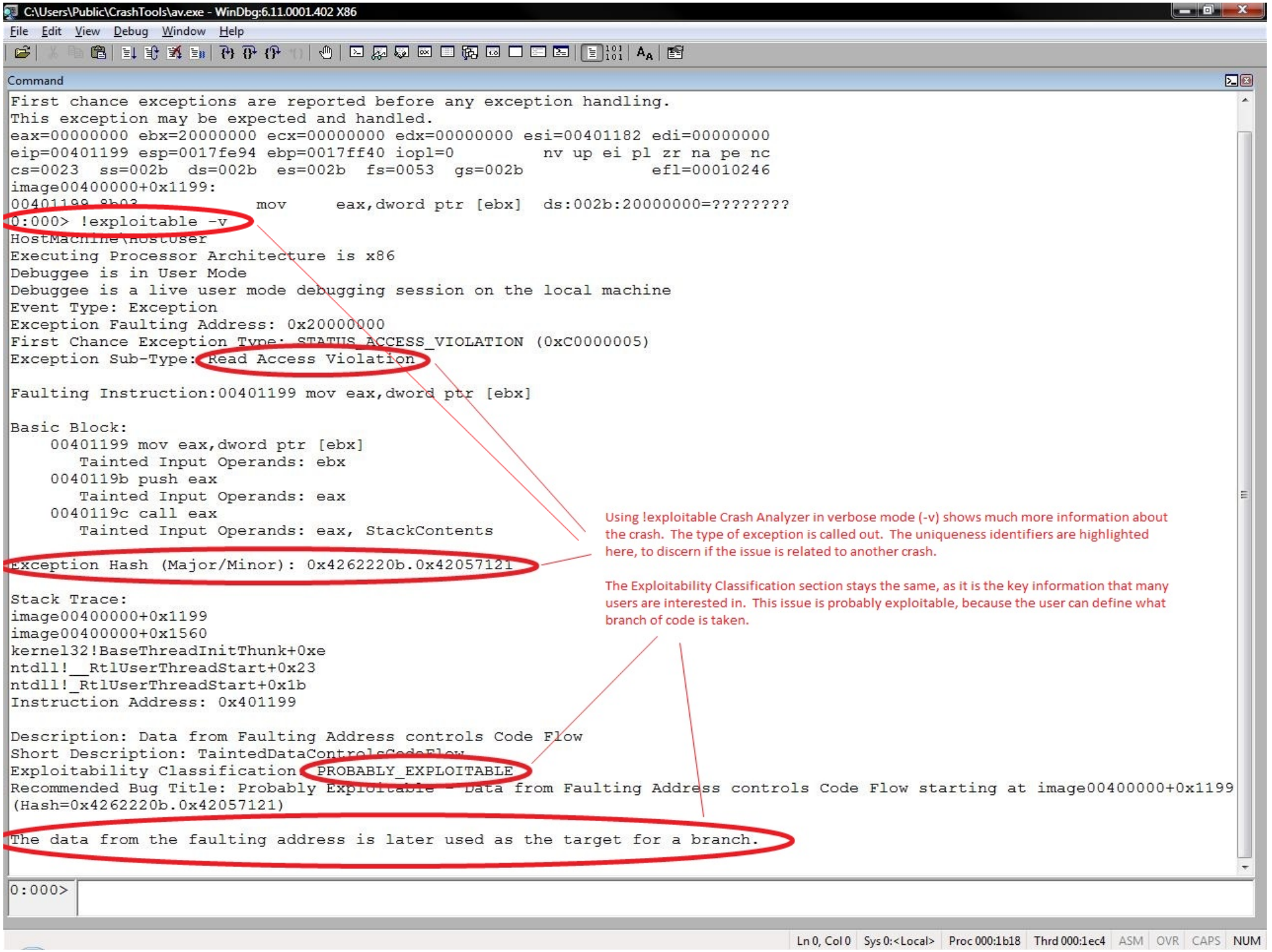
- Fundamental Assumption
  - All input operands in the faulting instruction are under the control of the attacker
- Limitations
  - The lightweight flow analysis and the above assumption can result in over-assessing risk
  - Registers are not aliased by value
    - Even if EAX and ECX were set to the same value before the faulting instruction, we will not consider them equivalently tainted
    - Changes in registers don't affect special cases
      - EBP-10 is a pseudo-register, unaffected by changes to EBP itself

Trustworthy Computing

!exploitable Crash Analyzer

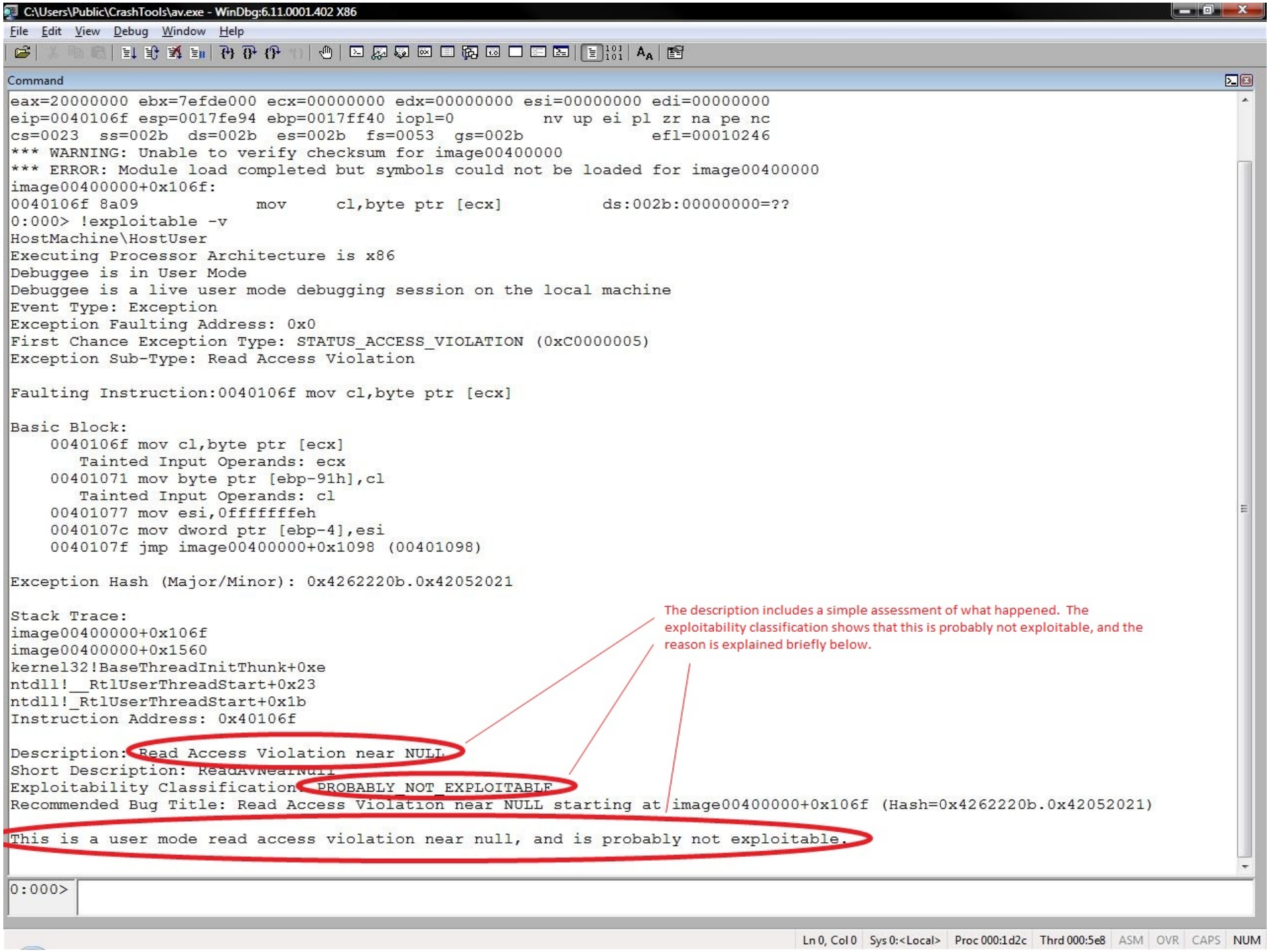
Walkthrough

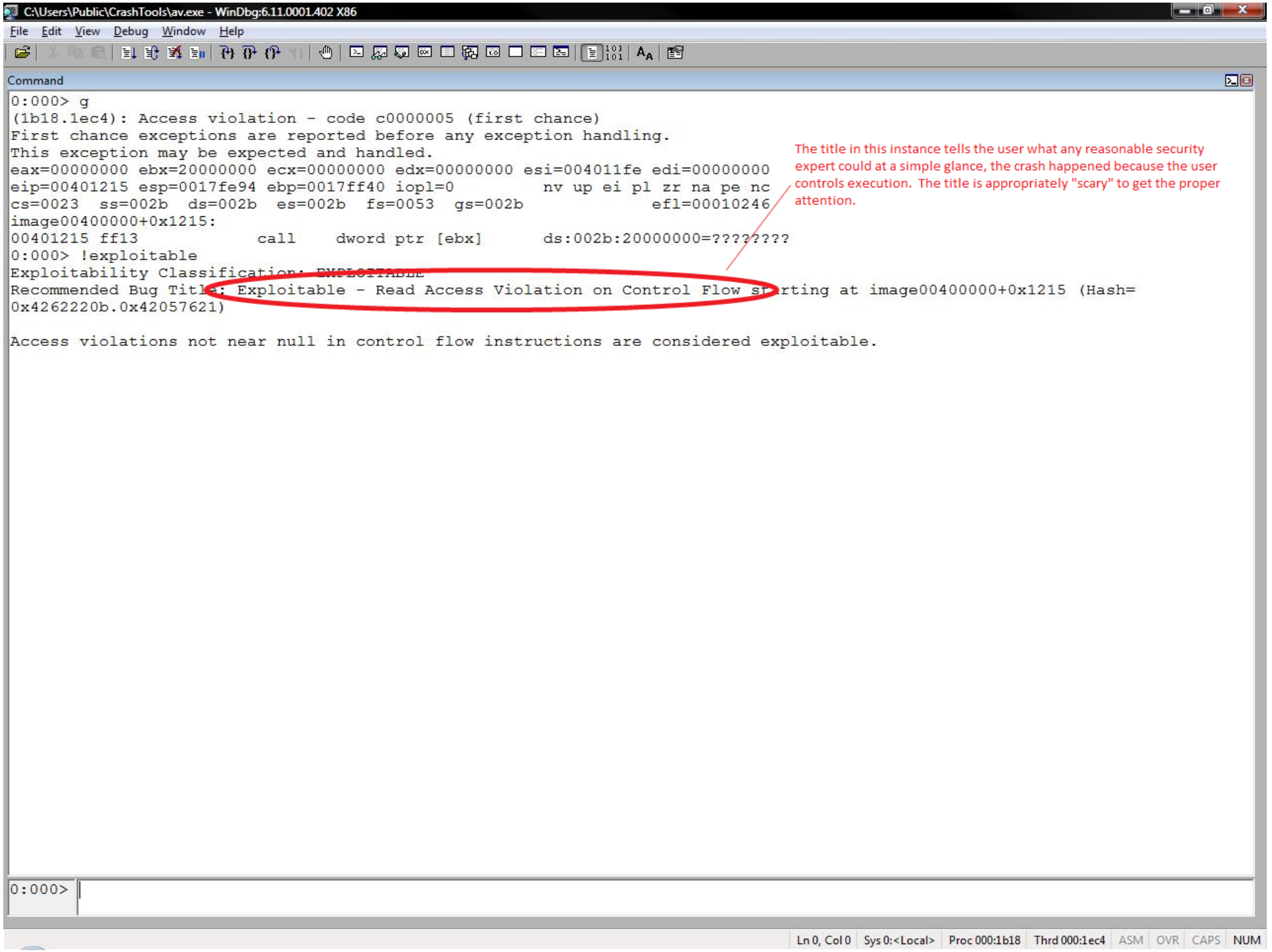

C:\Users\Public\CrashTools\av.exe - WinDbg:6.11.0001.402 X86
File Edit View Debug Window Help
Command
0:000> g
(1d2c.5e8): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000000 ebx=20000000 ecx=0000000c edx=00000000 esi=20000000 edi=0017feb0
eip=00401145 esp=0017fe94 ebp=0017ff40 iopl=0 nv up ei pl zr na pe nc
cs=0023 ss=002b ds=002b es=002b fs=0053 gs=002b efl=00010246
image00400000+0x1145:
00401145 f3a5 rep movs dword ptr es:[edi],dword ptr [esi]
0:000> !exploitable
Exploitability Classification: PROBABLY_EXPLOITABLE
Recommended Bug Title: Probably Exploitable - Read Access Violation on Block Data Move starting at image00400000+0x1145
(Hash=0x4262220b.0x42057021)
This is a read access violation in a block data move, and is therefore classified as probably exploitable.
!exploitable is run against a crash. The exploitability classification is set as Probably Exploitable, and an explanation is offered below.
0:000>
Ln 0, Col 0 Sys 0:<Local> Proc 000:1d2c Thrd 000:5e8 ASM OVR CAPS NUM



Using !exploitable Crash Analyzer in verbose mode (-v) shows much more information about the crash. The type of exception is called out. The uniqueness identifiers are highlighted here, to discern if the issue is related to another crash.

The Exploitability Classification section stays the same, as it is the key information that many users are interested in. This issue is probably exploitable, because the user can define what branch of code is taken.


C:\Users\Public\CrashTools\av.exe - WinDbg:6.11.0001.402 X86
File Edit View Debug Window Help
Command
eax=20000000 ebx=7efde000 ecx=00000000 edx=00000000 esi=00000000 edi=00000000
eip=0040106f esp=0017fe94 ebp=0017ff40 iopl=0 nv up ei pl zr na pe nc
cs=0023 ss=002b ds=002b es=002b fs=0053 gs=002b efl=00010246
*** WARNING: Unable to verify checksum for image00400000
*** ERROR: Module load completed but symbols could not be loaded for image00400000
image00400000+0x106f:
0040106f 8a09 mov cl,byte ptr [ecx] ds:002b:00000000=??
0:000> !exploitable -v
HostMachine\HostUser
Executing Processor Architecture is x86
Debuggee is in User Mode
Debuggee is a live user mode debugging session on the local machine
Event Type: Exception
Exception Faulting Address: 0x0
First Chance Exception Type: STATUS_ACCESS_VIOLATION (0xC0000005)
Exception Sub-Type: Read Access Violation
Faulting Instruction:0040106f mov cl,byte ptr [ecx]
Basic Block:
0040106f mov cl,byte ptr [ecx]
Tainted Input Operands: ecx
00401071 mov byte ptr [ebp-91h],cl
Tainted Input Operands: cl
00401077 mov esi,0fffffffeh
0040107c mov dword ptr [ebp-4],esi
0040107f jmp image00400000+0x1098 (00401098)
Exception Hash (Major/Minor): 0x4262220b.0x42052021
Stack Trace:
image00400000+0x106f
image00400000+0x1560
kernel32!BaseThreadInitThunk+0xe
ntdll!__RtlUserThreadStart+0x23
ntdll!_RtlUserThreadStart+0x1b
Instruction Address: 0x40106f
Description: Read Access Violation near NULL
Short Description: ReadAvNearNull
Exploitability Classification: PROBABLY_NOT_EXPLOITABLE
Recommended Bug Title: Read Access Violation near NULL starting at image00400000+0x106f (Hash=0x4262220b.0x42052021)
This is a user mode read access violation near null, and is probably not exploitable.
0:000>
The description includes a simple assessment of what happened. The exploitability classification shows that this is probably not exploitable, and the reason is explained briefly below.
Ln 0, Col 0 Sys 0:<Local> Proc 000:1d2c Thrd 000:5e8 ASM OVR CAPS NUM


C:\Users\Public\CrashTools\av.exe - WinDbg:6.11.0001.402 X86
File Edit View Debug Window Help
Command
0:000> g
(1b18.1ec4): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000000 ebx=20000000 ecx=00000000 edx=00000000 esi=004011fe edi=00000000
eip=00401215 esp=0017fe94 ebp=0017ff40 iopl=0 nv up ei pl zr na pe nc
cs=0023 ss=002b ds=002b es=002b fs=0053 gs=002b efl=00010246
image00400000+0x1215:
00401215 ff13 call dword ptr [ebx] ds:002b:20000000=????????
0:000> !exploitable
Exploitability Classification: EXPLOITABLE
Recommended Bug Title: Exploitable - Read Access Violation on Control Flow starting at image00400000+0x1215 (Hash=
0x4262220b.0x42057621)
Access violations not near null in control flow instructions are considered exploitable.
The title in this instance tells the user what any reasonable security expert could at a simple glance, the crash happened because the user controls execution. The title is appropriately "scary" to get the proper attention.
0:000>
Ln 0, Col 0
Sys 0:<Local>
Proc 000:1b18
Thrd 000:1ec4
ASM
OVR
CAPS
NUM

Who Benefits from !exploitable?

- !exploitable Crash Analyzer helps 3rd party software Developers and Testers working on Microsoft® platforms to manage their workload
  - Developers and Testers don't have to be security experts in order to identify many security issues
  - Can identify and categorize crashes with security implications quickly
  - Helps to prioritize work based on exploitability of crashes
    - "Exploitable" Elevation of Privilege bug may need immediate attention
    - "Probably Not Exploitable" Divide by Zero bug is likely a lower priority
  - Decreases the amount of time needed to analyze crashes for exploitability
- Security Eco System
  - Helps standardize exploitability reporting within companies and across the Security Ecosystem

Use Inside of Microsoft

- Used by Security Tools for crash categorization and bucketization
- Used by Developers, Testers and Support Engineers to evaluate the implications of crashes on products in development

Tool Tip: Include !exploitable output when reporting vulnerabilities

HOW MUCH IS ENOUGH?

When to stop fuzzing?

- An arbitrary number of iterations?
  - 100K, 250K, 500K, 1M?
- When you stop finding bugs?
  - How long to wait before your first bug?
  - When will you find your next bug?
- Some complex formula?
  - Parser complexity
  - File size
  - Code coverage

Case Study: Windows Vista® Fuzzing

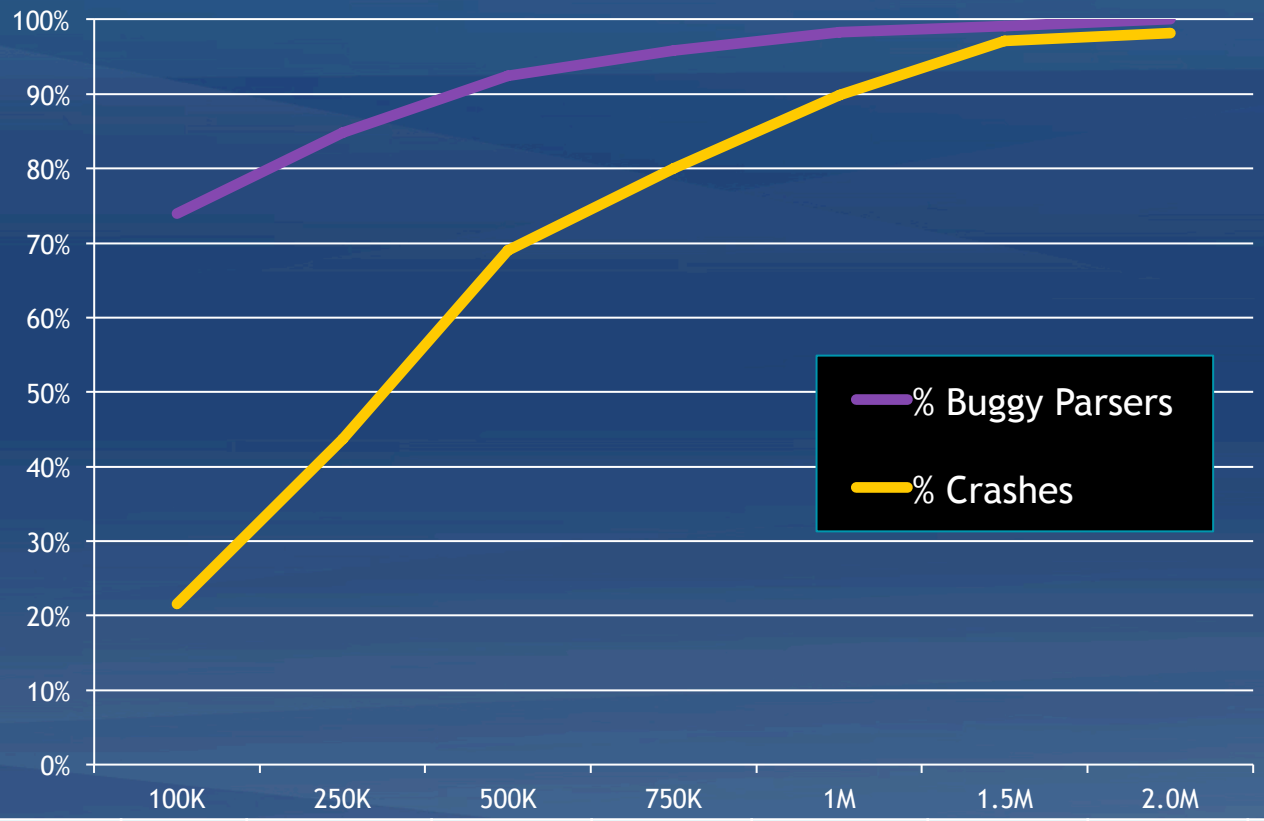
- Fuzzed from September ‘05 to November ’06
- 250+ file parsers fuzzed
- Bugs found and fixed
- 350M iterations total

- So, fuzz a 1 million iterations and fix 1-2 bugs?
  - Perhaps, but that might not be practical

Prelude to Windows Vista Statistical View

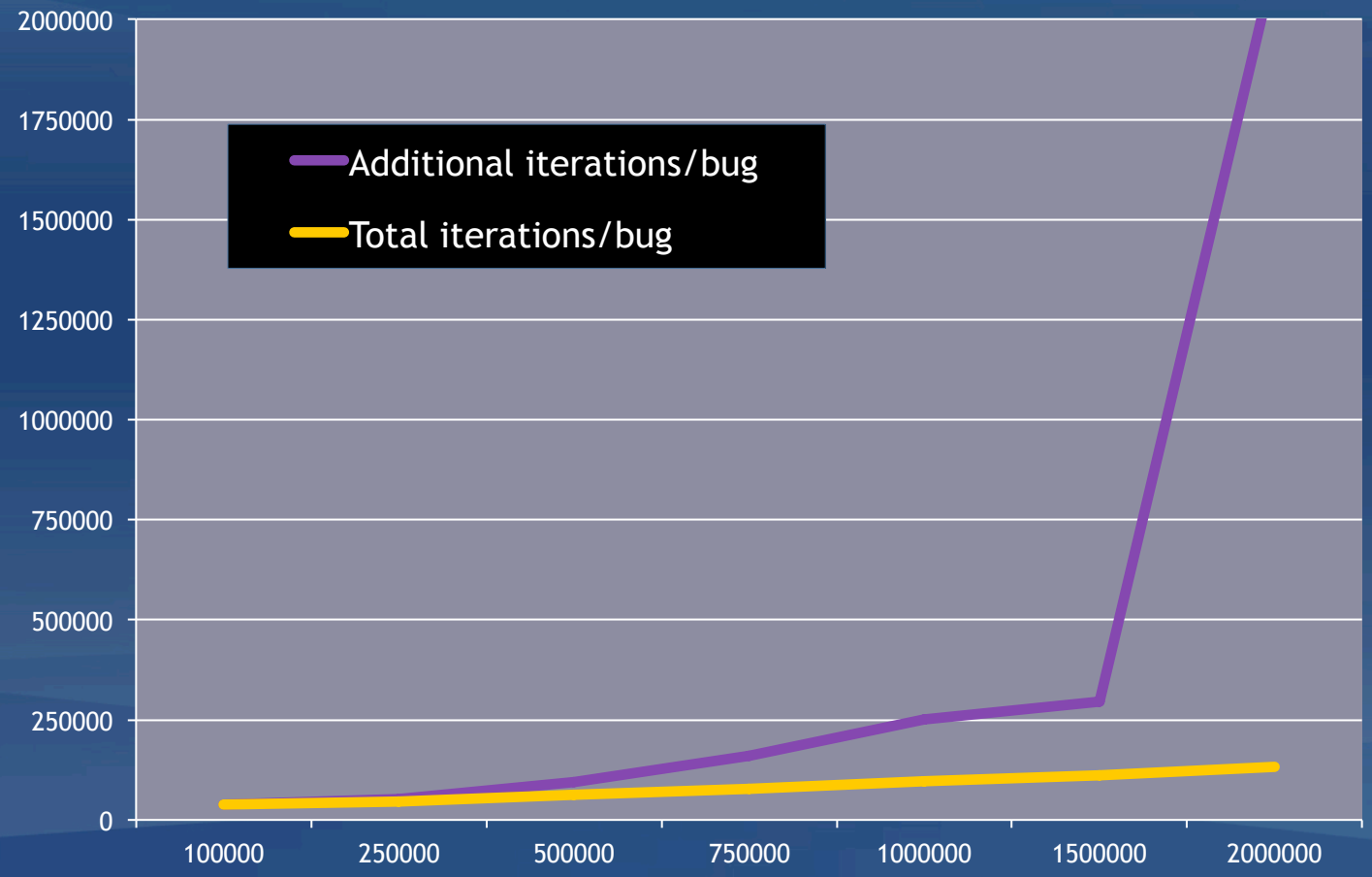
- Definitions
  - Parser: product code that interprets file data
  - Bug: reproduced failure in parser with distinct cause
  - Iteration: loading one fuzzed file into parser
  - Crash: automatically detected failure, grouped on distinct call stack
  - Buggy parser: parser containing at least one crash found in Vista
- Counts:
  - Parsers: 250+
  - Iterations: 350 million
  - Buggy Parsers: 120

Windows Vista Stats: Effectiveness at various counts



	100K	250K	500K	750K	1M	1.5M	2.0M	Full Run
% Buggy Parsers	74%	85%	92%	96%	98%	99%	100%	100%
% Crashes	22%	44%	69%	80%	90%	97%	98%	100%

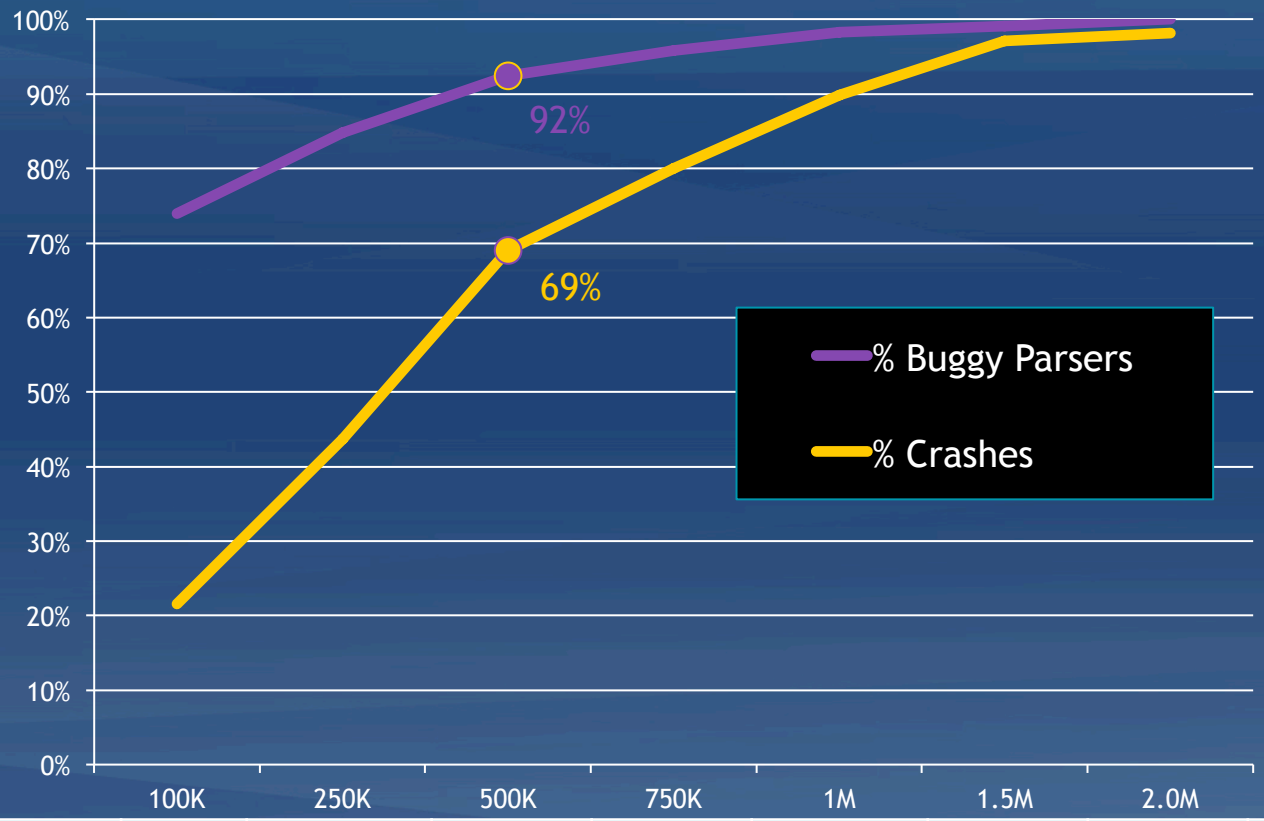
More Windows Vista Stats: Fuzzing Cost per crash



Windows Vista Fuzzing Conclusions

- Fuzz at least 500K for any exposed formats
  - Optimized templates can double the effectiveness
- KEEP FUZZING if you are finding bugs
  - Wait a minimum of 250K since your last new crash
- Consider ending points based on data, such as:
  - Number of crashes
  - Iterations since last crash
  - Code coverage
  - Code complexity
  - Complex statistical models
- When you have exhausted 1 fuzzer (more or less) try another method

Wait, 500K doesn't seem enough!



	100K	250K	**500K**	750K	1M	1.5M	2.0M	Full Run
% Buggy Parsers	74%	85%	**92%**	96%	98%	99%	100%	100%
% Crashes	22%	44%	**69%**	80%	90%	97%	98%	100%

Add the sliding window

	500K+0	500K+250K Δ	500K+500K Δ
% Buggy Parsers	92%	92%	92%
% Crashes	69%	91.8%	92.4%

- 500K gives high confidence that the parser is fairly solid if no crashes are found (90+%)
- 250K sliding window gives 20+% boost in bug finds
- But, an additional 250K window only gives <1% boost
- Is it still better to fuzz longer?
  - Certainly, fuzzing longer with many approaches is ideal
  - This gives a reasonable practical baseline.

Applying Our Learning

- Windows 7® File Fuzzing Effort
  - >10B iterations total
  - Fuzzers were more mature
  - Multiple types of fuzzers were used
  - 400+ file parsers fuzzed
  - Interestingly, about the same number of issues found and fixed
- Compared to Windows Vista
  - 250+ file parsers fuzzed
  - 350M iterations total
  - A number of issues found and fixed
- Fuzzing got better, bugs stayed the ~same
  - It takes MORE EFFORT to find crashes with fuzzing on an application as bugs are fixed

WRAP-UP

Jason Shirk
October 17, 2008

Conclusions – A Practical Guide to Fuzzing

- Invest up front in choosing your approach
  - Identify targets
  - Choose the best tools
  - Choose optimal inputs (Template Reduction)
- Diversify
  - Consider a mix of fuzzing tools and approaches
- Use !exploitable Crash Analyzer
  - Reduces triage time
  - Highlights important security issues quickly

Conclusions (cont.)

- Fix bugs aggressively
  - If there are too many bugs, consider removing the parser
- Fuzz long enough
  - At least 500K for reasonable assurance
  - Keep fuzzing while you are still finding new bugs
    - At least 250K Since the last bug
- Refuzz after fixes to ensure a quality release
- Mature the fuzzing efforts over time, don’t stagnate

Thanks!

- Michael Eddington (Leviathan)
- Microsoft - Adel Abouchaev, Dustin Duran, Tom Gallagher, Damian Hasse, Rob Hensing, Shawn Hernan, Vassilii Khachaturov, Scott Lambert, Michael Levin, Dan Margolis, Nachi Nagappan, Peter Oehlert, Andy Renk, Hassan Sultan, Gavin Thomas, Chris Walker, Dave Weinstein, Lars Opstad, Eric Douglas

Links

- For more information on Microsoft's Security Science and !exploitable Crash Analyzer, please visit:
  - http://www.microsoft.com/security/**msec**/
- And the Security Research & Defense (SRD) blog:
  - http://blogs.technet.com/srd

QUESTIONS?

Microsoft®

Your potential. Our passion.™